Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

SEBASTIAN JOSEF SCHMID
MASTER THESIS

A MODEL OF CODE REVIEW PRACTICES

A QUALITATIVE MULTIPLE CASE STUDY ON CODE REVIEW IN OPEN SOURCE

Submitted on 14 May 2019

Supervisor: Michael Dorner, Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 14 May 2019

License

Erlangen, 14 May 2019

Abstract

Code review is nowadays viewed as a crucial part of every software project. Benefits, efficiency and effectivity are therefore often the center of attention when looking at these processes. Beside this, the question arises how in a setting like open source software development code review is even formally defined and expected to be done. This includes topics like who is involved in a code review, what criteria are stated to accept or reject a possible change and who has the final say about the introduction of a change into the code base. In this thesis, we use a multiple case study approach to study six different open source software projects to gather the core expectations about the practices of their respective way of doing code review. We come to the conclusion that a common model can be described on how code review is expected to be done in open source software development.

Contents

1 Introduction

The overall goal in which this thesis wants to participate, can be defined in a very concise way: how is code review expected to be done in different open source software (OSS) projects and what are the similarities or differences between them? Of course this is a very broad topic, but this thesis wants to help to answer it by analyzing the formal settings of such code reviews. If different developers want to work together on a big software project, they need techniques to assure quality and interoperability between their work, if they want their project to survive and be actively used by a community. Without it things would get rather complicated to maintain with a growing code base - simply because only rarely one developer alone can know everything about every aspect of code.
This means that rules have to be stated and often these kind of rules are one of the first ones which are presented to interested developers who want to take part in such a project. It is not only important to know how the code works and how it looks like, but also what requirements the process behind a contribution to the code has. In the end, a contribution has to be accepted to make it to the code base. So if one wants to contribute, the developer has to know who is responsible, who will possibly review the change, what might benefit or harm the success of a proposed piece of code or what kind of change is considered important at all. These points are all part of a formal framework which has nothing to do with writing code itself, but rather defining the work between developers and aiming to make their lives and work better and more productive. The goal of this thesis is therefore to study this framework - and thus contribute to the overall goal.
Of course, different OSS projects have different expectations about this framework and how they want to work together. Different kinds of hierarchy and fields of responsibility come with various structures of how a contribution process and its code review could be organized. Due to this, it is highly interesting, how various projects handle this process, what they have in common and what might separate them. This is another goal of this thesis.
So in the end, there are two goals: on the one hand studying the formal process of code reviews of several chosen OSS projects and on the other hand analyzing, where these practices are standing compared to each other. This is done by performing a multiple case study.

2 Research Chapter

2.1 Introduction

Software is a product that can change very quickly and can be adjusted to almost any means that requirements demand. Therefore it is most important to check changes and adjustments with attention. Over time, means of checking code have been introduced to help developers with their task on writing useful code and preventing or finding errors. An example for a method from the past is the so called code inspection, a very formal way of reviewing code line by line in a group, which is reported to detect errors very efficiently, but also needs lots of preparation and organization (Galin, 2018). Today developers use easier and less formal approaches to review code changes which do not need a significant amount of preparation and are faster to carry out (Bacchelli & Bird, 2013).

But not only in closed industrial settings or in enterprises software is produced and during this process code reviews applied. Also in OSS projects, which have an equal interest of writing good software, code reviews are a mean of checking contributions, maintain quality and manage changes. These processes of course have to be defined by the respective authorities of these projects and considered by everyone who wants to take part in such projects (Prokop, 2010). The OSS communities therefore have to deal with these frameworks of code reviews.

The reasons and expectations behind doing code reviews are versatile and still subject of current research (Bacchelli & Bird, 2013). Initially defects, bugs and other errors should be caught before they can be introduced into the code base. Code should ideally only improve if it is changed. But also other positive side effects can occur. Not only is the whole code quality increased by applying code reviews, e.g. by enforcing a certain common coding standard or style guide, but also life between developers is influenced. Team awareness and productivity, especially in commercial settings, are touched by interpersonal expectations which are generated by code reviews: in the end, a developer does not want to be embarrassed by having errors pointed out in submitted code, so the efforts of passing the code review on first try by writing extra good code are increased.
Also effects of transparency occur, because it is much more difficult to make

quick-and-dirty fixes or change the direction of code development in a certain direction if other developers also have to give their approval. Lastly, code review comes with interesting passive benefits which can not only improve the code, but also the author's skills. By actively discussing and being exposed to the work of others, surely a reviewer will get to known new ways of thinking and coding. Likewise an author is expected to deal with the reviewer's comments and think about made errors and proposed solutions. This is something that goes beyond the pure amount of external links and references that are provided and used by authors to improve their code. It is a process that is steadily and subconscious. Nevertheless it is a very important aspect and benefit of code reviews, which can lead on the one hand to personal training and growth of individual developers, but also on the other hand to a better, more homogeneous development team with almost equal experience.
Alternative solutions result partly from defect detection and partly from knowledge sharing. If a review is not only seen as a limited time where code has to be examined, but also has space for discussion and group-based thinking and problem solving, as it has been observed especially for OSS projects from Rigby and Bird (2013), a review can quickly evolve from pure defect finding to think tank like work groups. However the original task of performing reviews may not be lost out of sight, but surely the chance of getting together and improving the code while one is at it, is something desirable.

As it can be seen, the expected benefits behind code reviews are promising and useful, so an OSS project has enough reason to state rules for a code review process. This means a theoretical framework is issued on how code review is expected to be done. The consequence for contributors and developers is that they not only have to deal with the code and its properties itself, but also with processes and paths which have to be taken to get their ideas accepted and committed to the code base. So if one wants to contribute, the developer has to know who is responsible, who will possibly review the change, what might benefit or harm the success of a proposed piece of code or what kind of change is considered important at all. These information are all part of a formal framework which has nothing to do with writing code itself, but rather defining the work between developers and aiming to make their lives and work better and more productive. The contribution of this thesis is thus an analysis of this theoretical code review framework by performing a multiple case study on several OSS projects, extracting the below stated key information according to our research questions and comparing their practices to each other.

2.2 Related Work

Code review practices, benefits and processes are all topics of current research. They are not only considered for settings in open source software development, but also in industrial and academic settings.

In their work, Bacchelli and Bird (2013) researched the modern, lightweight form of code review (in contrast to classic formalized code inspection) with focus on expectations, outcomes and challenges. By observing and conducting interviews as wells as surveys with employees of Microsoft, they concluded that the expected benefits of code review are far more than only finding defects, despite this being the number one reason why it is done. It also promises positive effects on the overall coding standard, solution finding and improvement of the development team, e.g. by knowledge transfer and team awareness just by being exposed to the work of others. These expectations give a good reason why code review is done at all, and why even in an industrial setting the invest of performing reviews is done.
Similarly, Rigby and Bird (2013) compared the practices of industrial projects, e.g. of Google and Microsoft, with OSS projects, e.g. Linux and KDE, to find out the respective parameters of peer review (such as review intervals or number of involved reviewers) deviate among these projects. They found that the similarities among these projects are quite significant. Especially in OSS development, code review has to follow a "lightweight, flexible process" where preferably small changes are reviewed as quickly as possible before they are committed. Also, modern reviews changed in general from pure defect finding sessions to group problem solving.
The survey of Bahamdain (2015) discusses quality assurance in OSS development, especially the general framework and processes in defect detection. They state that after a bug is detected, possibly reported and verified, the code for solving the bug is on one hand permanently reviewed by the author, which is called self-review by them. On the other hand, before the code can be handed in, it needs to be reviewed by another developer such that no new defect is introduced and the solution for the former defect is in fact provided. They explain that the group of core developers and co-developers would provide in this context high quality code, care about fixing defects and reviewing code contributions.
In a case study of the Apache Server project, Rigby, German and Storey (2008) study the peer review process for review-then-commit and commit-then-review to determine its parameters and compare them to formal code inspections. Here they focused on parameters like how many reviewers would respond to a review or how long a review takes. By examining review threads, mailing lists and commit logs, they found that most changes are reviewed by two individuals (despite the Apache policy asking for three core-group members for a review) and over

80% of reviews are reviewed in less than 3.8 days. They state that in the case of Apache, the "early, frequent reviews of small, independent, complete contributions conducted asynchronously by a potentially large, but actually small, group of self-selected experts leads to an efficient and effective peer review technique."
Rigby (2011) extends the work of Rigby et al. (2008) by examining six OSS projects in terms of archival records of reviews and semi-structured interviews with developers with focus on review practices. He comes to the conclusion that reviews are often chosen by reviewers according to interest and skill, which also defines their involvement. In terms of why patches tend to be rejected, the interviewees state that mostly purely technical reason or violations of the project's scope are the issue. The latter often tends to spark lengthy, unproductive discussions which threaten the effectivity of code reviews. All in all, OSS development regards code reviews as most important quality assurance practice.
The work of McIntosh, Kamei, Adams and Hassan (2014) focused on software quality in OSS in dependence of code review coverage and participation. In a multiple case study of Qt, VTk and ITK projects, the authors found that software quality might suffer because of a lack of participation and discussion in code review, which can lead possibly more post-release defects. Furthermore they suggested that high code review coverage rates do not necessarily hinder software from becoming defect-prone, which leads them to the assumption that other influences are present.
Baum, Liskin, Niklas and Schneider (2016) formulated a classification scheme to describe code review processes in industrial settings. To get their structure, they studied reviews in 19 companies by interviewing software engineering experts and augmented their findings with a semi-structured literature review. They conclude that despite all interviewees having a common idea of code review, they deviate in the details. Their scheme concentrates on describing facets like process embedding (e.g. tools or RTC and CTR), reviewers (e.g. amount and population), checking (e.g. interaction and roles), feedback (e.g. type of communication) and overarching facets (e.g. tool specialization). Their results can be used to describe review processes accordingly, but do not make general statements about the framework.
Sadowski, Söderberg, Church, Sipko and Bacchelli (2018) researched the code review process at Google to investigate practices, motivations as well as satisfaction and challenges with it. Not only is code readability and maintainability more important than defect finding, also the review process focuses on being lightweight and flexible and values code ownership by assigning only one single reviewer. Despite mostly reviewing only small and iterative changes, developers view code review as necessary and valuable. Hindrances in the process itself are often due to geographical and organizational distance, social gradient between the involved programmers and mismatches about the review subject (e.g. design reviews) or the context of the change, for instance why it was introduced in the first place. In general, Sadowski et al. observe that the bulk of reviewed changes

are small, have at most one reviewer and no comments beside an approval to commit. About 70% of the changes are committed less than 24 hours after the review process has been started.
Rigby and Storey (2011) investigate interaction between programmers and mechanisms in broadcast based peer review used in OSS projects. They found that participation as a reviewer mostly depends on the personal interest in a change and also that patches which fail to generate such, are rather ignored. Depending on the nature of a patch, technical or in dependence of the project's objectives, different reviewers take interest and part, whereas there are no specific roles assigned to them. They also find reviewers' personality and the general tone in a review to take a major role, e.g. a negative contribution would be a rude one or one that is too specific to be relevant in general. Furthermore they investigate mailing lists for reviewers in the Linux project and find that, while it is allowed to add specific reviewers, still others contribute which leads to advantages from broadcast mailing lists as well as expert reviews.

Based on the related works given above, we can see that code review, and especially in OSS, is a field which attracted a good amount of attention. Often questions are asked about the effectiveness of reviews, which parameters in general are influencing the code review, who takes part and what makes a code review process a good one. But seldom the question arises, what the projects behind the code review process even expect, what they see fitting as a change or what is considered something to make it accepted or rejected. These questions, which will be defined in detail in chapter 2.3, are the ones we want to answer.

2.3 Research Questions

In this section we define our research questions. As we explained in our motivation in the introduction, our overall goal is to analyze the theoretical framework of how code review is expected to be done in OSS projects. To answer this, we want to study the following research questions in relation to OSS projects:

RQ1: "Which roles are involved in the review?"

Many developers interact together in an OSS project on their code. On the one side is of course the author who wants to make a contribution or a change, but who else is involved in the code review process? And if they are involved, on which basis - maybe voluntarily or just because it is their respective responsibility? Or have they even been picked by the author? With this question, we take a look at all other roles in the code review who are involved.

RQ2: "Which roles accept or reject a change?"

On some point in time the review might come to a positive end and no more objections or concerns are left to deal with. But is the change now immediately merged into the code base? Is it accepted without any further thought after the review and applied? Or is there another instance, another check, who has the final say about if a change is accepted or rejected, even if it passed the code review? This question analyzes hints about the possible outcome of code reviews. We examine what is stated about the actual possibility of implementation of the change and whose decision it might be.

RQ3: "What are acceptance and rejection criteria for a change?"

Changes are thoughtfully reviewed to maintain a certain quality in the code base, such that it is necessary to apply standards of expectations in the review. But which kind of criteria are usually applied when a change is proposed by an author? What kind of advice is given by the OSS project to meet these criteria and make it more likely to be accepted? And what is considered a reason for rejection from the beginning on? Here we analyze what is stated from the official sources what might increase or decrease the chance of submitting successfully code and passing the review.

RQ4: "When does code review happen in the development process?"

Code review is a crucial part of software development and has direct influence on what finally makes it to the code and what not. Merging an unstable or error-prone piece of code too early might introduce therefore a risk for the rest of the software, so the question arises how it is dealt in general with new code: is the review done before the commit, called review-then-commit (RTC) or after, called commit-then-review (CTR), as defined by Rigby (2011)? Are there maybe any special situations, where exceptions might be applied? Here we analyze if something is specifically stated about these situations and how the timeline of the code process is defined normally.

RQ5: "What is the coverage of reviewed changes?"

Code submissions might contain lots of changes, but do really all of them have to be reviewed? Or are there some kinds of changes which are excluded per se, and if yes, for what reason? With this question, we take a look at the general review policy of OSS projects of what has to be reviewed at all.

In the following chapters of this thesis, we present the results related to these research questions. We will come back to discuss these and their results in chapter 2.6.

2.4 Research Approach

To study how code review is expected to be done in OSS projects and their similarities and differences, we performed a multiple case study as defined by Yin (2014).

2.4.1 Multiple case study design

According to Yin (2014), a case study can be used to study a phenomenon in its natural, real world context. Our overall question in this context is how code review is expected to be performed in the context of OSS, therefore a case study is a proper way to answer this question. Furthermore, we do not need any control over the specific "behavior" we want to research, in this case the way of performing code review: we can not and do not want to control how OSS projects review their code changes, so a case study is still the best choice as compared to a controlled experiment. Lastly, the subject we are focusing on, the code review, is a contemporary topic and not entirely historical. Yin defines "historical" as events where no direct observation could be done anymore, but in the case of active, alive OSS projects as the ones we have chosen (cf. chapter 2.5), this definition does not apply.
The technique we are using is defined as a cross-case analysis (Yin, 2014). A single case study in this context would produce only a singular contribution about how a specific OSS projects would handle its code review and what expectations are given. But because we want to differentiate between several OSS projects to get an insight in their similarities and differences, we need several cases we can study. Thus we need a multiple case study with different defined cases, where we can analyze each ones theoretical frame work on code review, and after that a way to compare these.

2.4.2 Case definition

The projects we want to study have to fulfill several criteria such that they can be considered a valid case for our multiple case study:
First of all, they need to be an OSS project, meaning that it is a project which permits non-exclusive commercial exploitation, availability of the work's source code and derivations of the work itself. (St. Laurent, 2004)
Second, we want to investigate code review as a contemporary topic and so we need OSS projects that are still under development, although we allow them to already have released versions. As long as there is still active development ongoing we consider the project to be alive. Therefore projects which are completely released and entirely finished would only provide entirely historical data.
Third, open collaboration and code review must be implemented in these projects.

Obviously it does not make sense to investigate an OSS project's framework if none exists. Because of this we ask our projects to have such kind of processes defined and of course also accessible to us and potential new developers.
Fourth, to get a well-rounded profile, we want to define our cases as mix of literal and theoretical replications (Yin, 2014) of each other, where we expect a common ground in terms of the code review framework. Two variations are regarded in this context, the one is the domain, which is covered by the project's scope, e.g. the project is an operating system or a browser. The other variation is the age of the OSS project, where we want to get a range of older, more established projects with potentially more sophisticated processes, and newer projects, which could give insight into more recent processes and new approaches in code review. Our main interest in selecting cases as theoretical replications is, that we want to cover a good basis of OSS projects in general and not a single area of it, such that our multiple case study is well-rounded. Literal replications give us the opportunity to directly compare projects for common aspects.

All in all, we define our cases as contemporary, varied OSS projects with active development and code review process, to get a comprehensive insight into the common practices and expectations of Open Source Software code review.

2.4.3 Case selection

According to our definition of possible cases, many projects would be possible candidates. As a mature and successful project to start with, we consider the Linux Kernel, which is already running since 1992. From there on, we get FreeBSD as a literal replication, and the newer projects listed as theoretical replications with variation across age and covered domain. The complete list of selected cases is given in Table 2.1.

Table 2.1: Selected cases for multiple case study

Project	Age	Domain
Linux Kernel	1992	Operating System
FreeBSD	1993	Operating System
LLVM	2000	Compiler
Chromium	2008	Browser
OpenStack	2010	Cloud computing
React	2013	GUI framework

We explain which data sources were used, the process of data collection and how they connect to the research questions in chapter 2.5.

2.4.4 Analysis methods

The logical connection between the gathered data concerning the review processes and the results has been made according to Mayring (2015). We applied qualitative content analysis methods to the data sources with the goal to gather relevant information concerning our research questions, which is defined as structural content analysis.

For this, we pre-defined categories on our built knowledge base around the topic, which was derived from the related works on this topics (cf. chapter 2.2) and the research questions per se. Then we applied these categories via coding to the gathered material of the single cases and refined the categories in two iterations to get to detailed assignments of material parts to emerging answers for the research questions.

After coding the material, we determined the characteristics of the several cases as it is suggested by Mayring for content-oriented structuring: the extracted, relevant material has been paraphrased and then summarized according to categories. From there on, we interpreted the found data of each single case and compared them to the other cases with focus on common or differentiating aspects. Based on this, we built our theory of what OSS projects expect as a framework for their code review processes.

As a tool for gathering, categorizing, coding and comparing the data of the single cases, we used MAXQDA2018 (2018).

2.5 Used Data Sources

The research questions in chapter 2.3 ask for investigations with focus on the theoretical framework of how code review is expected to be done in OSS projects. The definition of the cases and which ones we want to study have already been stated in 2.4.3. To get appropriate data for our research, we need valid data sources which are within in the bounds of the case study. Yin (2014) defines six different "sources of evidence" for a case study: documentations, archival records, interviews, direct observations, participant observation and physical artifacts.
Because of the limited time in which this work was written, we decided not to gather evidence in form of interviews, direct observations or participant observations, although this direct interaction with developers and reviewers might yield very deep insight into the code review process itself and how the participants understand the expectations of their projects.
Physical artifacts are exemplary defined by Yin as "technological device, a tool or instrument, a work of art, or some other physical evidence". None of these categories can be applied to the expectations of code review, therefore we did not consider them as source of evidence for this research.
Archival records would concern quantitative evaluations of code reviews, e.g. the data base of a reviewing tool, which could give a good insight of how review is actually done. This kind of source is often used by other researchers (e.g. in the work of Rigby (2011)) who want to understand the effectiveness of reviews, but in our case this does not apply. We want to investigate what is expected rather than what is actually done, therefore this source is of little use for us and not considered.
Finally, documentations yield the information we want to examine. The official online wikis and web representations of the selected OSS projects often state rules and hints for new developers about how the development process works and what they have to bear in mind. This also concerns the review process, where often is stated under which conditions patches or changes can be handed in, and also sometimes mentioned what might be a benefit or what should not be done. We consider these official statements, given in the online sources, as explicit declaration in terms of what is expected on how code review has to be done and in what kind of theoretical framework it is embedded. This is the main source of evidence we have used for our research, but also linked sites and documentation from these official websites where appropriate. References to the respective OSS sources, which have been examined, are given in appendix A.
After getting to the official sources, the data collection and analysis methods stated in chapter 2.4.4 have been applied. This means that the pre-defined codes have been applied to the data and refined, afterwards extracted, paraphrased and then summarized according to category concerning the research questions.

In regards of difficulties during the data collection process, we can report no restrictions in terms of accessibility of the wikis or the linked sites. In the end, OSS projects want to propagate their processes to new and interested developers. Therefore they have to make their requirements publicly accessible and clear how things work. Unfortunately, this information is often neither available in one spot nor written in a manner that states specifically what is the gist. Examples, code snippets or vague hints in the text can distort the core messages and different terminologies for the same things exist among the various projects. By the means of the distributed nature of wikis and beginner's instructions, as well as possibly bloated text, it might be possible that not all details in every single case could have been gathered. But because we are investigating several OSS projects as cross-cases and compare them to each other, we are confident that the respective hints of evidence of the single cases complement each other to give a wholesome picture.
Furthermore, it is seldom stated what exactly is considered as beneficial or hindering in a code review, but mere instructions on how a the process is working and what is expected to be done or included. From this basis on, we had to interpret the possible effects of such statements on the answers to our research question. For instance, if it is stated in connection to code reviews that the styleguide has to be respected in every contribution to the code base, we interpret this point as it being beneficial for the change to be accepted. That means in this example, if the styleguide is respected, the code change is more likely to be accepted than rejected by the reviewer.
Lastly, it has to be considered that the online sources are used as they are presented officially by the OSS projects. We assume in our data collection process that all these sources are up to date, well maintained and represent the truth and very own intention of these projects. If this would be of course not the case that, for example a wiki would be outdated by far and deviate from the current expectations of its project, the gathered data would be invalid. To the best of our knowledge, all the used data sources stated in appendix A are up to date.

The following part of the chapter presents our gathered data of each single case which was stated in chapter 2.4.3. The results of the cross-case analysis and answers to our research question will be given in chapter 2.6.

2.5.1 Linux Kernel

For Linux Kernel it is custom that each patch has to be reviewed for quality and that it provides desired changes to the code base. This means there is a strict review-then-commit (RTC) practice for normal patches, which is summarized by the uncompromising statement:

> *'Patches do not go directly from the developer's keyboard into the mainline kernel. There is, instead, a somewhat involved (if somewhat informal) process designed to ensure that each patch is reviewed for quality and that each patch implements a change which is desirable to have in the mainline. This process can happen quickly for minor fixes, or, in the case of large and controversial changes, go on for years. Much developer frustration comes from a lack of understanding of this process or from attempts to circumvent it.'* - kernel.org

The only exemption from this process might be fixes for exploitable security issues, which have to be deployed as quickly as possible. These are not handed in over the regular public mailing lists for review, but instead given to a specified email address and reviewed afterwards.
In terms of who is involved in the review process beside the contributor, we identify the reviewers in early stages, who respond according to mailing lists where potential changes are posted and their personal interest. Also maintainers, who are defined as code owners and experts on the respective area which is changed, and other developers who work on this code are informed and expected to review the change. Especially for bug fixes, the original owners have to be contacted as well.
After a review is finished and accepted, the final decision has still to be made if the change will be merged into the code base, which does not automatically apply. Instead, an approval of an appointed authority has to obtained. In general this authority is defined as follows:

> *'There is exactly one person who can merge patches into the mainline kernel repository: Linus Torvalds.'* - kernel.org

Linus Torvalds theoretically acts as central authority, which can decide about all patches, but in reality he does not. He cannot manage all changes on his own, therefore there exists a so called lieutenant system of appointed top-level maintainers who represent him in their respective subsystem of the kernel. The chain leads further down in a "chain of trust" to other maintainers with their respective field, such that these experts are expected to only accept changes which conform to the stated standards of the projects. If these lower positions accept a patch, it is normally automatically approved by the higher positions in the hierarchy because of this trust, and then merged into the code base. As a consequence they are distinctively called gatekeepers. Therefore this whole systems relies on

a distributed system of authorities with mutual trust, acting together as a board which controls the possible patches.
Criteria for the approval or rejection of a change are given in the Linux Kernel wiki in form of hints and checklists. They tell developers how they can get their submissions to be accepted more quickly. Next to points like technical compliance (e.g. the patch is tested and has no compiler warnings or errors) and following the style guide, it is also crucial to document all changes in the code. Interesting for acceptance of the patch into the kernel is also that a documented endorsement by reviewers, who are known to be specialists on the subject, increase the chance of getting approved overall. Social interaction and communication between author and reviewer do also play a major role, where politeness and patience are considered as very positive. Miscommunication and undesired behavior, like ignoring answers or trying to take shortcuts in this process, are regarded as severe violation. To give an example of how sincere these inter-personal rules are for Linux from a reviewer's view, we would like to quote the subsystem maintainer Greg Kroah-Hartman as an example.

> *'Here's some very easy and simple steps that you can follow to ensure that you make a Linux kernel subsystem maintainer mad enough to never want to read your patches again: [...]*
> *- Send patches that ignore the well documented and established coding style rules.*
> *- After having the aforementioned coding style rules pointed out to them, continue to send patches which ignore them. '* - kroah.com

Beside these rules for basic interaction, it is also asked for that patch emails for reviews are easy to read and quick to process. It is explicitly stated that if e.g. a change is given as a MIME attachment, it would be impossible to comment the code directly. Also it would take more time to process (especially for Linus Torvalds), what on the other hand would decrease the chance for getting accepted.

2.5.2 FreeBSD

The official wiki of FreeBSD states that all ”non-trivial” changes have to be reviewed before they can be committed to the code base (RTC), especially for parts where the programmer is inexperienced with. However, if a developer is working on a bug in an abandoned area of the system, commits can be made just like that. The same applies for working on ones own code. For the case that code is changed, which belongs to a certain other maintainer, the FreeBSD wiki defines a coarse exemption from code reviews:

> *'Only if the maintainer does not respond for an unacceptable period of time, to several emails, will it be acceptable to commit changes without review by the maintainer. However, it is suggested that you try to have the changes reviewed by someone else if at all possible'* - freebsd.org

> *'If queries go unanswered or the committer otherwise indicates a lack of interest in the area affected, go ahead and commit it. [...] If there is any doubt about a commit for any reason at all, have it reviewed before committing. Better to have it flamed then and there rather than when it is part of the repository.'* - freebsd.org

This does not mean that no review will take place, just because no one obvious is there to care about it, but rather that it is postponed. A review will therefore take place.
Reviewers involved in this process are picked by the author via Phabricator, the used reviewing tool. First of all the contributors are asked to refer to code owners, defined as original authors of certain changed code pieces, or the maintainers who took over. Also other committers to the same code are considered as potential reviewers and lastly volunteers, which can be added or chosen by the author. All in all the reviewer should at least be an expert to be really able to understand the change and review it properly. This serves as a possibility of last resort.

> *'In some cases, no subject-matter expert may be available. In those cases, a review by an experienced developer is sufficient when coupled with appropriate testing.'* - freebsd.org

Beside being a first position as reviewers, maintainers also function as owners and responsible authority for tracking changed code. Only if a maintainer agrees, the code can be contributed. The only exemption is the above stated case that no answer is given for a too long period of time. Beside this, code can also have a group of developers act as equal maintainers and not only a single responsible. New committers also have a mentor who has to give an approval for potential changes to the FreeBSD repository and acts as a guide.
Criteria for the approval of a change and a positive code review are given in

statements concerning what to provide for a review. Formal points like handing in a defect-free code, documentation of the change (e.g. proper description and title) and a manageable workload, meaning several small changes which are fast to review, are expected to be done. Also the declared style guide has to be taken into account. Coding style and readability are topics which are quite endorsed by FreeBSD and are referenced from the official wiki to the website of Julio Meroh as an explanation to the FreeBSD review process culture.

> *'Code reviews exist to give someone else a chance to catch bugs in your code; to question your implementation in places where things could be done differently; to make sure your design is easy to read and understand (because they will have to understand it to do a review!); and to point out style inconsistencies. '* - julio.meroh.net

As already mentioned, additionally the endorsement of a mentor is obligatory for new committers.
What is explicitly emphasized as "strongly discouraged" it the usage of advertising clauses and new licenses in new code. Legal matters like these shall be avoided and if they are nevertheless applied, it has to be inevitably approved by the core development team.

2.5.3 LLVM

According to the LLVM developer policy, all changes by all developers have to be reviewed before they can be committed (RTC), especially if significant changes are introduced. Only small changes or patches in areas where the author owns the code can be treated the other way round, meaning commit and then review (CTR). In the same manner trusted contributors are allowed to commit their work first and let it review afterwards. Nevertheless all code is required to be reviewed at some point, hence there are no exceptions from this rule. The combination of this process is called their formula for success by LLVM.

> *'The LLVM Project relies on two features of its process to maintain rapid development in addition to the high quality of its source base: the combination of code review plus post-commit review for trusted maintainers. Having both is a great way for the project to take advantage of the fact that most people do the right thing most of the time, and only commit patches without pre-commit review when they are confident they are right.'* - llvm.org

The reviewing process is organized via Phabricator, where the author is asked to pick one or two suitable reviewers. LLVM recommends to pick and include the code owner of the respective area of code in each case. The role of the code owner is exactly defined as the one who has to make sure that in the assigned are all changes are reviewed. And even if the owner would not have enough time to

review any changes, it is still in the owner's responsibility to make sure it is still done. Furthermore, other contributors to the same code piece are recommended as suitable reviewers because of their work in the same area. In general anyone could act as a reviewer and also code review from anyone who is interested is welcome. Of course the self-selected reviewers by the author are only asked to review the patch, but not required to participate. They can also refuse to be picked.
All changes require on the one hand the approval of the reviewers from the LLVM community who review the code. Only if they actively give their agreement in the end, the change can be further processed. LLVM states in this context, that there exists nothing like a "silent approval" or a request for "active objections to the patch with deadline". The whole process is expected to be regarded as iterative and as a consequence needs this specific agreement in the end. The final approval of the change lies in the responsibility of persons with subversion write access.
Changes are required to have a clean and readable documentation, such that it can be clear what is tried to achieve with it. The same is valid for the coding style, which should always follow the style guide to a source that is "uniform and easy to follow". This is explicitly embraced. Also the provided code must be technically compliant to be accepted (e.g. no errors or warnings on compiling) and should include a testcase. In the same manner the provided tests from LLVM have to be passed. If the commits fail to comply to the technical and quality standards of LLVM, they are rejected, but the author is welcome rework the change and trying again. Regarding the nature of the change and the inter-social aspects, developers are asked to ensure beforehand that bigger contributions are discussed with the community.

> *'The design of LLVM is carefully controlled to ensure that all the pieces fit together well and are as consistent as possible. If you plan to make a major change to the way LLVM works or want to add a major new extension, it is a good idea to get consensus with the development community before you start working on it.'* - llvm.org

Especially for big changes, it is desirable that code changes are as small as possible and split up into several patches which build on each other. This reduces the workload for each code review, makes it faster and according to LLVM increases the chance that a reviewer would take a quick look at it. In the same fashion this tends to give code pieces a higher coding standard and reduces the chance to get a negative feedback from reviewers.
As mentioned above, violations against the style guide or non-compliant technical errors are clear reasons to reject a change. Also it is explicitly stated that including legal matters, which conflict with the terms of the LLVM license, are a reason to exclude and reject the contribution, too. This includes adding confidentiality or non-disclosure notes.

2.5.4 Chromium

The development process of Chromium defines as default way that first a review has to take place and then code can be committed (RTC) via the commit queue. Beside this there are also defined steps where a commit can be made with a review afterwards, called "To Be Reviewed" (TBR) in this context, which is the same as a commit-then-review (CTR). It is again emphasized that nevertheless a review has to take place and cannot be omitted. There are two common reasons where CTR can be applied, where the first one is to revert previous changes that broke some important functionality, to get the system running again. The second stated reason for TBR are emergencies which assume not enough time for proper review. After code has been directly committed in this case, still the respective code owners have to be informed and a review has to take place as defined in TBR.
Regarding the part of what has to be reviewed, it is given several times that all changes have to be reviewed, whether RTC or CTR is applied. The intention behind this is simple:

'Code reviews are a central part of developing high-quality code for Chromium. All changes must be reviewed.' - chromium.googlesource.com

In the review involved are several roles. On one hand any committer can be appointed as reviewer by the author and participate. On the other hand at least one owner for each changed directory has to be included and give approval to the changes in the review. In general it is embraced to choose reviewers who are familiar with the code parts that are touched by the author or at least know the low-level code parts.
Approval to a code change is given by the reviewers, when they are satisfied with the patch, which is an incremental process. Beside the agreement of the common reviewers, especially the approval of the code owners of the sections is needed. Only if their mandatory assent is given, a contribution can be introduced into their respective area. This means, if a patch touches several directories, all owners have to agree in the end or the patch cannot be applied.
In this context it should be mentioned that the whole review process of Chromium and the guideline for authors is captioned by the name "Respectful Changes". Authors are asked to document their code well and give suiting descriptions with the problem as well as what they are changing and why they are changing it. This ensures readability and saves time. Following the sytle guide is required and using respective tools to validate ones code is encouraged, which would save again time and show respect to the reviewers. The usage of Clang's formatting is stated as something that should be always accepted by reviewer. In terms of respect, authors are also reminded that code reviews are centered around code and not the person behind it. A polite way of interaction is encouraged:

'Code reviews are in large part about having others watch your back. Don't hesitate to say "Thank you" once the review is completed. Additionally, if you're new to code reviews, take a few moments to reflect on what went well or didn't.' - chromium.googlesource.com

This is also encountered again in the demand to make small changes and hand them in as a series or separate patches, to "spare your reviewer time or cognitive load". Big patches on the other hand are unlikely to get reviewed quickly. Also, authors have to make sure that their code is ready for review at all, which means that it can compile without errors and passes all tests. Chromium ties this is again to respect of reviewers who would expect code to be provided in such a manner.

2.5.5 OpenStack

In OpenStack a contribution can only be made, if a review has been successfully carried out and approved (RTC). This also means that all changes and contributions to the project have to be reviewed. The approval for a change normally requires agreement from non-core reviewers as well as mandatory from core reviewers, but it is sub-project dependent what was agreed upon. After that approval is given, the change is merged automatically into the code base by Gerrit, the used tool. This is the standard way for changes, but for certain circumstances, like trivial patches, this process can deviate. Although the approval of core reviewers can be dropped or reduced, it still is an RTC process.
Involvement in the review process includes on one hand the already mentioned core reviewers, whose expertise is crucial to the whole process. Beside them any developer in the project can give comments or be a reviewer and give a vote for the change, although it is not stated how the reviewers are picked.
The approval process for a change relies on a voting system, which gives regular reviewers the chance to give a +1 or -1 to a change during review. Positive votes indicate agreement with the change as it is, negative votes the opposite. Additional to this, core reviewers can give a +2 or -2 vote to indicate their opinion in a same manner. Nevertheless of the sum of votes, which represents the general consent of reviewers, at least one +2 vote is necessary for a change to be approved and merged. Mind that a +2 vote and two +1 votes are not the same. The authority of the core reviewers is very strong in this context and it is convention that two +2 votes are needed. Their role as experts is underlined as only merited members of the project's community can be appointed:

'A core reviewer is a member of the OpenStack community that has volunteered to dedicate time to reviews for a specific project. To become a core reviewer, the interested person is required to have provided enough reviews to the project in order to demonstrate the understanding of the project structure, goals and policies.' - docs.openstack.org

Once the change is reviewed and accepted by the reviewers and core team, it is set to a respective state and added to the job queue. It will run through several tests automatically and, if they are passed, be merged to the project's code base. OpenStack presents a list of several points to "keep in mind when doing code review", which is addressed to developers. It is emphasized beforehand that through reviews the social norms and development processes of OpenStack could be learned. Beside this, review guidelines have also been stated which aim to help identify issues, make the review easier and guide the interaction between developers. An important point is documentation, which includes e.g. comments in the code or in the commit message. Clear labels about what and where things are fixed or at least a clear title and summary have to be given. Typos in important key words of a commit message are for example considered as valid reason to reject a change.

> *'For example, if there is a patch submitted which a reviewer cannot fully understand because there are changes that aren't documented in the commit message or code documentation, this is a good time to issue a negative score. Patches need to be clear in their commit message and documentation.'* - docs.openstack.org

As clear as the documentation should be, also the code has to look like: the style guide of the sub-project has to be followed, and if none exists the OpenStack's general style guide is valid. This requirement is strongly related to the demand to give "pythonic" code. The code should "look like the code around it, to make the code more uniform and easier to read". Readability and maintainability are both emphasized again in the general guidelines for reviews. Of course, technical compliance is also demanded, e.g. that the code should built and pass all test without errors, introduce no new bugs or that technical standards in the project are followed. Examples for this are the logging standard or the support of python2. For general interaction, the guidelines summarize the gist in the short sentence:

> *'Review is a conversation that works best when it flows back and forth.'* - docs.openstack.org

Reviewers and submitters alike are addressed in this part of the guideline, but submitters in particular are asked to be responsive to the comments of reviewers and address their requests. Equally, reduced complexity in changes is preferred, where patches focus only on solving one problem at a time. If the patch is too big or complex, the change will be rejected and the contributor asked to split it up or rework it to get multiple reviews for it.

2.5.6 React

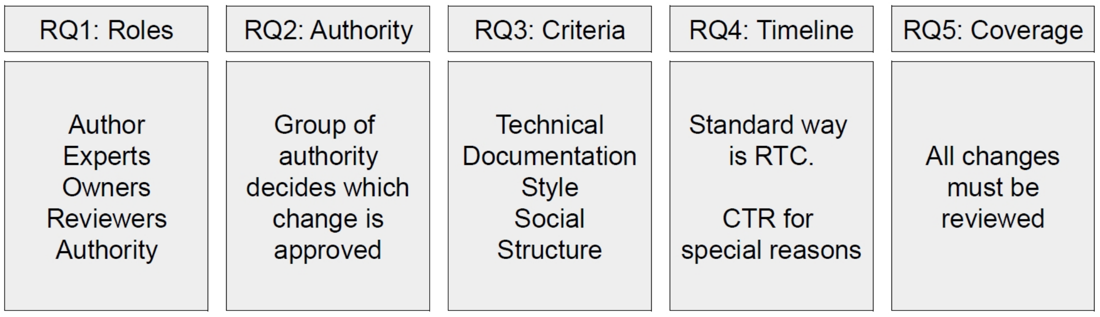
Although the review process documentation of React is sparse, it is apparent in the contribution guide that their process follows an review-then-commit process (RTC). The pull-request, which is initiated and send out by the author, will be reviewed by the core team of React. They will then decide about the status of the change. Either it will be accepted by them and therefore merged, changes requested or rejected.
Crucial to the submission is that the Contributor License Agreement for React is accepted once - without it no pull request can be accepted. For the code itself, React has stated a list of points to be checked before a changes are handed in. What is considered important, is to follow the style guide and of course to provide error free code. For the style guide criteria, React provides an automatic code formatter name 'Prettier' which should be used - also to get a good and readable coding style. Technical compliance is also required, e.g. by asking the author to include tests for fixed bugs. Distinctly marked is the point that React will not consider violations against their core principles.

> *'We wrote this document so that you have a better idea of how we decide what React does and what React doesn't do, and what our development philosophy is like. While we are excited to see community contributions, we are not likely to choose a path that violates one or more of these principles.'* - reactjs.org

These core points include things like interoperability, stability or the basic composition of React. All in all, the fundamental style of React may not be violated at all. Issues like these or impacts like performance issues and features that inflict difficulties on building future applications, are reasons to reject a submitted change, according to Alpert, software manager and core team member at React.

2.6 A model of code review practices

After we have presented the respective cases and their expectations to code review, we want to consider these frameworks and their differences and similarities to build our model of code review practices. This happens with respect to our research questions, which we have defined in chapter 2.3, in the form of a multiple case study. The given topics, concepts and expectations emerged by applying the qualitative content analysis according to Mayring to the material, as stated in chapter 2.4.4. A short overview over all findings with respect to their research questions is given in figure 2.1 and explained later on in detail.



RQ1: Roles	RQ2: Authority	RQ3: Criteria	RQ4: Timeline	RQ5: Coverage
Author Experts Owners Reviewers Authority	Group of authority decides which change is approved	Technical Documentation Style Social Structure	Standard way is RTC. CTR for special reasons	All changes must be reviewed

Figure 2.1: Overview over findings for each research question

RQ1: "Which roles are involved in the review?"

Beside the author who wants to submit a change, there is a whole variety of persons involved in the review process. Coarsely they can be split up into four groups, which interact on several levels and can overlap even in persons and responsibilities: experts, owners, reviewers, and authorities. Not all projects state these roles explicitly, but they make use of the responsibilities and parts they represent in the review process.

Author
The author is the role which initiates the change and hands in code for review. While all other stated roles can take directly part in the process by reviewing the code, the author cannot be a reviewer. Nevertheless the impact of the role is given by the general influence on the review, e.g. by selecting the reviewers or the manner how the code is presented to them. In the end it is the author's work which is considered during review.

Experts
Experts are developers who have certain knowledge in some area of code, either because they have already committed to that part, have appropriate expertise

about the functionality or are simply part of the maintainers of this area. They are endorsed as reviewers by all projects and often stated together with owners or authorities. Their judgment of the change can be considered crucial, as in OpenStack, where merited core reviewers can block a change completely.

Owners
Owners act as roles which are responsible for the respective area and piece of code which is altered by the change. They can also be experts and extend their role by the fact that they are solely responsible for all that matters in their piece of code. Often they are included as reviewers (e.g. in Chromium mandatory, in FreeBSD and LLVM only recommended) or have to be at least informed about the change, because they have to ensure that a review takes place (as in LLVM). In any case they have to give their approval to the change.

Reviewers
Reviewers are the complete group of developers who review the changes and also form a superordinate group for experts, because they are normally reviewers, and owners, although they don't have to actively review but still can choose to do so. Reviewers can be self-selected by the author (e.g. experts), can get involved voluntarily because of their own interest (regular developers) or because it is their responsibility to do so (e.g. owners in Chromium). Self-selected reviewers are addressed by the author directly, for example via personal mail or via the respective tool which is used and then expected to take part in the review. Criteria for picking these reviewers are often given in a loose manner, but can be summarized to choose experts and owners. Other reviewers, which are not self-selected, can get active, because the change has to be announced via a special mailing list, like in Linux. Then everyone in this mailing list has a chance to act as reviewer and give comments. Other ways include that fixed group of reviewers exists, which will always review all changes without any influence of the author, like the core team in React.

Authorities
Authorities are persons which are not the respective owners or maintainers of a piece of code and not otherwise directly involved in the review itself, but still have to be considered in the process. While owners act as "local" authority and can take directly part as reviewers or even as expert, authorities only consider the more abstract level of changes, f.i. if the philosophy or goals of the project are touched by the change. While in the end all projects ensure that only changes are merged which suit their goals, only the Linux project states Linus Torvalds and the lieutenant system as single authority which act in this context as last decision maker. The other projects distribute the authority of approving a change more

on the other, directly involved roles.

The common ground in the chosen OSS projects is therefore that review should be done by someone who is directly involved in the changed code, either through expertise or responsibility. However it is diverse how authors get their reviewers, either because they pick them or because the reviewers get active on their own, either voluntarily or by obligation.

RQ2: "Which roles accept or reject a change?"

Of course the reviewers, consisting of experts, owners and other developers, have to decide during the review, if a change can be accepted or not. But beside these, the change itself also has to be approved such that it can be merged into the code base. This is part of the authority role, which is also involved in the process.
The common ground is that in all cases the authority is held by a respective group of people, which decide if an approval can be given and act together as a board. This can be the core reviewers in OpenStack, the sum of owners in Chromium, the chain of trust in Linux, the core team in FreeBSD and React or the person responsible with subversion write access in LLVM. Normally one of these group members has then an area of responsibility, where such approval is given on behalf of the group. All in all, these authorities have also a common background: they are designated by merit and trust. First, it is very beneficial if someone is given such authority, if they have worked for some time for the project in a constructive way, and second, they are given trust in terms of decision making about whether or not a new change can be merged into the mainline.
The main difference is however, how far this authority goes: whereas in Linux subsystem maintainers have far more rights in terms of merging and feature decisions, core reviewers in OpenStack have just their focus on reviewing changes. This if of course inherent to the role behind the authority - if it is only defined as a role to review and not to decide about the future of the project, one cannot expect it to have more rights.

RQ3: "What are acceptance and rejection criteria for a change?"

Five big common points emerge from the stated criteria of the OSS projects, about what is expected as beneficial for a code review. These criteria are technical criterion, documentation criterion, style criterion, social criterion and contribution structure.

Technical criterion
Not only code without any defects, bugs or mistakes is considered as technical compliant. Also if current or correct libraries are used, the compilation has

neither warnings nor errors, and all provided tests are passed, then code is considered as functional and compliant with the technical standards of the project. This is a core expectation of all projects and without it, no change will be accepted in the code base. Strong rejection criteria are changes that contain defects or lead to regressions, instabilities, and performance issues.

Documentation criterion
As important as correct code is considered, is also the need to document it properly. This applies on the one hand for documentation in the code, like comments, on the other hand also for commit documentation and messages about the patch itself. Certain parts like the title, a description of the features and what code parts are touched are also essential as documentation for the commit itself. The benefit is simple, because if reviewers can easily read what a change should do and what the code does, it is far more easy to understand it and to give appropriate feedback. Code which is not or only very sparsely documented tends to be very hard to understand and therefore is likely to be rejected until everything has been explained properly.

Style criterion
Even if the code is functional and the documentation for the patch is done, the manner of the code itself is important to the project. The projects aim to get readable and maintainable code, such that the sources are easy to understand and have a uniform style. This is also ensured by the common request to follow the style guide and coding conventions. It is beneficial if the reviewers see that the change follows these requests and is easy to understand, although small, justified deviations are often allowed.

Social criterion
Beside the change itself, the general interaction with reviewers and other involved persons must be considered. OSS projects expect their authors to be aware of social norms and manners, especially during reviews. This involves core traits like being polite and patient, and to support the reviewing process by being open to communication and discussing about the code. Self reflection and an open mind for the opinion of others help avoiding a deadlock. On the other hand, things like trying to work around the review process, being impatient or not responding to comments during review are not beneficial and also considered rude towards reviewers and the OSS project.

Contribution structure
What also has to be kept in mind is the workload which is generated by handing in a change. It is embraced that breaking up bigger patches into smaller parts

makes reviews faster and reviewers likelier to take a look at it. Also it is stated that smaller changes are easier to understand and therefore are more likely to be accepted. In short, if authors are aware how they structure their contribution into smaller workloads, they can influence the likelihood of their reviewers accepting these smaller patches.

The stated points are common to all selected OSS projects. Beside this, some projects also explicitly state that the introduction of non-applicable legal matters, f.i. new licenses or non-disclosure notices, are reasons to reject changes. On the other hand of course, the OSS's license has to be agreed to such that a contributor can even hand in code. These issues are also important but only rarely stated directly in relation to review criteria.

RQ4: "When does code review happen in the development process?"

All OSS projects implemented an review-then-commit process as standard way to commit a change to the code base. This means that an author has to give the change to one or more reviewers, potentially self-selected or involved by other circumstances, get their approval and only then it will be merged into repository. Beside this regular process, some projects have defined commit-then-review processes for special reasons. These include the "to-be-reviewed" (TBR) process of Chromium for emergency commits or reverting patches that have broken the core functionality, or less urgent reasons, like for very trivial changes, as it is defined by LLVM, or if simply too much time has passed and a request for reviews goes unanswered by the responsible maintainer, like FreeBSD rules. But in every case it is assured and explicitly explained, that even if the commit goes first, in every case a review will take place and nothing is left out.

RQ5: "What is the coverage of reviewed changes?"

The common consent of the chosen OSS projects is that all significant changes must go through a reviewing process at some point in time. The only part where the projects differ, is the definition of their minor exceptions of this process. E.g. if code is considered orphaned, FreeBSD states that a commit can take place without checking with other developers, or that only non-trivial changes should be reviewed. But nevertheless, it is stated often in all cases that all changes have to be reviewed, independent of RTC or CTR, emergency or incident. The reason for this is simple and has already been given as a citations from LLVM and Chromium in their respective cases: via reviews, the projects expect to maintain a high quality in their code, and to achieve this they defined their review processes.

2.7 Discussion

In this chapter, we discuss the limitations of our multiple case study as well as concerns which could arise. To address this, we use the four common quality criteria for case study research as defined by Yin (2014). These tests are construct validity, internal validity, external validity and reliability.

2.7.1 Construct validity

Construct validity tries to identify whether the correct operational measures for the studied concept have been applied. In our research this measure for studying the review model of OSS projects is that we gathered their published descriptions of their review processes and then analyzed them. All used data sources are directly related to the OSS projects and therefore represent insights into the definition of their processes. Concerns about the measure to get our results are therefore addressed by this fact, because our chain of evidence is established.
Furthermore we use multiple sources of evidence in our multiple case study. Although we were only able to use process documentation from online sources in our study, we have documentation of several different cases. These complement each other to give a wholesome of sources about the review processes in general. Data triangulation with respect to other sources, like interviews etc., could not be established. The reasons for this are stated in chapter 2.5.

2.7.2 Internal validity

Internal validity tests the causal relation between a studied effect and a possible explanation which is given by an explanatory study. The method we used is a multiple case study, which concentrates on analyzing and describing the framework in which OSS projects define their review process. With this we describe our code review model. Because our research is therefore purely descriptive, and not explanatory or causal, we do not address rival explanations or pattern matching. We study what is expected to be done in these review processes and do not try to explain why.
The gathered raw data needed interpretation to be evaluated during cross-case analysis, which means researcher bias might be possible. The risk of introducing own expectations towards the research and its results is reduced by using content-oriented structuring. This helps as analytic strategy to find emerging answers and topics based on the data itself and not mainly influenced by our assumptions.

2.7.3 External validity

External validity addresses the possibility to generalize our findings. Since our research is not based on a survey or other quantitative methods, we cannot use the argument of statistical generalization to generalize our findings to some kind of population where a respective sample would be picked from. Rather we use analytic generalization to justify the possibility to generalize our findings with respect to our replication logic. Since we chose our cases as literal and theoretical replications, we got well-rounded cases we could study. Furthermore, by applying a cross-case analysis to these cases and considering all cases with respect to each other, we get robust findings.

2.7.4 Reliability

Reliability ensures that the findings of this thesis can be repeated by another researcher with the same cases again. We ensure this by using MAXQDA2018 as a tool which saves our raw data and coding in a database, such that all used data and their analysis can be understood. Also the documentation of our used methods and processes is given in the methods section (chapter 2.4) of this thesis. Together with the used data sources and shown limitations in chapter 2.5, we are confident that our research can be repeated.

2.8 Future Works

After our research has been presented and its limitations have been discussed, we would like to take a look at possible future works.

In general we described a theoretical framework of code review practices which contains expectations of OSS projects. This does of course only describe how a process should ideally look like, but not necessarily how they are practiced in reality. A next step would therefore be to evaluate the model towards the processes and how they are realized by authors and reviewers. Interesting points in this would be differences and similarities between model and reality as well as potentially why they occur. The ratio of how these expectations are actually met could be an indicator if these processes are defined in a realistic and proper way or if they do not meet real world issues at all.

The review processes are defined as they are because of specific reasons, mostly because several benefits are expected. These expected benefits are already covered by current research, f.i. Bacchelli and Bird (2013). A new aspect would be to study how the defined framework of these processes influences these presumed benefits or quality in a positive or negative way. For example, changes are expected to fulfill the style criterion to make code readable, but on the other hand might force authors to change their own style in a drastic way. An interesting aspect of this point would be if therefore also the change's code quality is different as compared to no enforced style.

Another point to compare with this model would be the expectations of industrial software projects towards their code review processes. The circumstances of how code is written could differ very much in terms of potential reviewers, role distribution and acceptance criteria. Therefore it might be interesting to see how a potential model of industrial code review might look like and how it relates to the one of OSS. Another approach could involve comparisons to inner source techniques. And in the end, of course it is interesting to see why these differences or similarities occur between the several models.

2.9 Conclusion

In current research, code reviews are a topic that are studied often with respect to questions like efficiency, influencing parameters or general motivation and benefits. The formal framework, how a code review is expected to be done in OSS projects, has not been described up to now. Points like involved roles and their responsibilities, criteria for assessing changes, the timeline and coverage of the general process were treated by this thesis.

We performed a descriptive multiple case study among six alive OSS projects, which were chosen by us according to the stated criteria to get a well-rounded selection. To get to our findings, we first analyzed statements from online documentation of these projects about their formal definitions of their review processes and how they should be done. Then we took the results of these six projects and made a cross-case analysis to get common and separating points which answer our stated questions. Together, these answers build a theoretical model of code review processes in OSS projects. This includes findings like that the standard review process is an RTC process, or that beside technical, documentation and style criteria, also social aspects and the contribution structure play an influence, if a change is accepted or not.

Although case studies cannot be directly generalized, we are confident that by analytic generalization the built model from our research can be. Overall, the findings can be considered as robust, because of the number of cases and the complement of each other. This is achieved by cross-case analysis and the performed structural content analysis.
With this described model on OSS code review, further steps can be taken to use and refine it. The realizations of the review process could be checked, if it is implemented according to this model. Also, how the process or stated criteria for accepted or rejected changes influence the software quality. A comparison to other processes in industrial settings is likewise imaginable.

All in all, code review is a topic that already gets a lot of attention, but unfortunately, this attention is very concentrated on few aspects. Efficiency, effectivity and influences of special characteristics of review are often considered. The general structure, the framework, behind the processes is on the other hand just assumed to be there and existent. We believe that by researching this overall framework, other, new processes could also be defined and in the end the presumed influences better identified and validated and therefore new statements about code reviews made.

Appendix A Used Data Sources

Table 2.2 gives the respective references to the data sources for each single case.

Table 2.2: Data Sources for all cases

Case	**Accessed on**	**URL**
Linux Kernel	12.02.2019	https://www.kernel.org/doc/html/v4.14/process/2.Process.html
	12.02.2019	https://www.kernel.org/doc/html/v4.17/process/5.Posting.html
	12.02.2019	https://www.kernel.org/doc/html/v4.17/process/6.Followthrough.html
	13.02.2019	https://www.kernel.org/doc/html/v4.17/process/submitting-patches.html
	12.02.2019	https://www.kernel.org/doc/html/v4.17/process/submit-checklist.html
	13.02.2019	http://www.kroah.com/log/linux/maintainer.html
	13.02.2019	http://www.kroah.com/log/linux/maintainer-02.html
	13.02.2019	http://www.kroah.com/log/linux/maintainer-03.html
	13.02.2019	http://www.kroah.com/log/linux/maintainer-05.html
	13.02.2019	http://www.kroah.com/log/linux/maintainer-06.html
FreeBSD	18.02.2019	https://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/policies-maintainer.html
	18.02.2019	https://www.freebsd.org/doc/en/articles/committers-guide/conventions.html
	18.02.2019	https://www.freebsd.org/doc/en/articles/committers-guide/pre-commit-review.html
	18.02.2019	https://www.freebsd.org/doc/en/articles/committers-guide/pref-license.html
	18.02.2019	https://www.freebsd.org/doc/en/articles/committers-guide/developer.relations.html
	18.02.2019	https://wiki.freebsd.org/action/show/Phabricator?action=show&redirect=CodeReview
	25.02.2019	https://wiki.freebsd.org/BecomingACommitter
	25.02.2019	http://julio.meroh.net/2014/05/code-review-culture-meets-freebsd.html

LLVM	18.02.2019	https://llvm.org/docs/CodingStandards.html
	18.02.2019	https://llvm.org/docs/Contributing.html
	18.02.2019	https://llvm.org/docs/Phabricator.html
	18.02.2019	https://llvm.org/docs/DeveloperPolicy.html
Chromium	12.02.2019	https://www.chromium.org/developers/contributing-code
	12.02.2019	https://chromium.googlesource.com/chromium/src/+/master/docs/code_reviews.md
	12.02.2019	https://chromium.googlesource.com/chromium/src/+/master/docs/cl_respect.md
	12.02.2019	https://dev.chromium.org/developers/contributing-code/direct-commit
	12.02.2019	https://chromium.googlesource.com/chromium/src/+/master/styleguide/c++/c++.md
	12.02.2019	https://chromium.googlesource.com/chromium/src/+/master/styleguide/c++/c++-dos-and-donts.md
OpenStack	16.02.2019	https://docs.openstack.org/infra/manual/developers.html
	18.02.2019	https://docs.openstack.org/infra/manual/core.html
	18.02.2019	https://docs.openstack.org/project-team-guide/review-the-openstack-way.html
	18.02.2019	https://docs.openstack.org/project-team-guide/open-development.html
	18.02.2019	https://docs.openstack.org/openstack-ansible/latest/contributor/code-rules.html
React	25.02.2019	https://reactjs.org/docs/how-to-contribute.html
	25.02.2019	https://reactjs.org/docs/design-principles.html
	25.02.2019	https://www.youtube.com/watch?v=wUpPsEcGsg8

References

Bacchelli, A. & Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering* (pp. 712–721). ICSE '13. San Francisco, CA, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2486788.2486882

Bahamdain, S. S. (2015). Open source software (oss) quality assurance: A survey paper. (Vol. 56, pp. 459–464). The 10th International Conference on Future Networks and Communications (FNC 2015) / The 12th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2015) Affiliated Workshops. doi:https://doi.org/10.1016/j.procs.2015.07.236

Baum, T., Liskin, O., Niklas, K. & Schneider, K. (2016). A faceted classification scheme for change-based industrial code review processes. In *2016 ieee international conference on software quality, reliability and security (qrs)* (pp. 74–85). doi:10.1109/QRS.2016.19

Galin, D. (2018). *Software quality. concepts and practice*. Hoboken ; Hoboken ; Hoboken, NJ, USA: IEEE Computer Society ; IEEE Press ; Wiley.

MAXQDA2018. (2018). Qualitative Datenanalyse mit MAXQDA - Software für Win und macOS - MAXQDA - the Art of Data Analysis. Retrieved from https://www.maxqda.de/

Mayring, P. (2015). *Qualitative Inhaltsanalyse: Grundlagen und Techniken*. Weinheim: Beltz Pädagogik.

McIntosh, S., Kamei, Y., Adams, B. & Hassan, A. E. (2014). The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th working conference on mining software repositories* (pp. 192–201). MSR 2014. doi:10.1145/2597073.2597076

Prokop, M. (2010). *Open-Source-Projektmanagement. Softwareentwicklung von der Idee zur Marktreife*. changes. München: Open Source Press.

Rigby, P. C. (2011). *Understanding open source software peer review: Review processes, parameters and statistical models, and underlying behaviours and mechanisms*. ProQuest Dissertations and Theses.

Rigby, P. C. & Bird, C. (2013). Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th joint meeting on foundations of* *software engineering* (pp. 202–212). ESEC/FSE 2013. doi:10.1145/2491411.2491444

Rigby, P. C., German, D. M. & Storey, M.-A. (2008). Open source software peer review practices: A case study of the apache server. In *Proceedings of the 30th international conference on software engineering* (pp. 541–550). ICSE '08. doi:10.1145/1368088.1368162

Rigby, P. C. & Storey, M.-A. (2011). Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd international conference on software engineering* (pp. 541–550). ICSE '11. doi:10.1145/1985793.1985867

Sadowski, C., Söderberg, E., Church, L., Sipko, M. & Bacchelli, A. (2018). Modern code review: A case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice* (pp. 181–190). ICSE-SEIP '18. doi:10.1145/3183519.3183525

St. Laurent, A. M. (2004). *Understanding open source and free software licensing - guide to navigating licensing issues in existing and new software*. O'Reilly.

Yin, R. K. (2014). *Case study research - design and methods*. Sage.